

国际化作家平台加载性能分析

问题

番茄小说原创国家化作家平台的 Slardar 页面性能数据显示，页面整体 FCP 都偏大（平均 >3500ms），发文页的 FCP 相比其他页面则更大（平均 > 4500ms）。

时间	指标	基线	占比			波动	均值	2022-11-09	2022-11-08	2022-11-07	2022-11-06	2022-11-05	2022-11-04	2022-11-03	2022-11-02	2022-11-01
			良好	可改进	欠佳											
2022-11-10 09:01:11	首页 FCP	2500,4200		100%		-10%+8%	3467	3523	3502	3543	3665	3738	3499	3414	3520	3409
	作品列表 FCP	2500,4200		100%		-11%+9%	3698	3597	3839	3817	3946	4025	3745	3642	3829	3616
	章节列表 FCP	2500,4200		100%		-8%+5%	3485	3654	3422	3456	3469	3503	3530	3515	3554	3626
	编辑器 FCP	2500,4200		14%	86%	-10%+15%	4562	4433	4556	4484	4211	4599	5027	4351	4627	4708

为了更好的用户体验，需要分析影响页面性能的真实原因并进行优化。

一：发文页 FCP 时间长分析

分析

初步分析

发文页与其他页面的主要区别是使用了编辑器的 SDK，且初步分析发文页面的 JS 文件体积名列前几名，所以是否是因为编辑器导致文件打包体积较大而导致 FCP 较慢？

Show chunks:

- ✓ All (13.67 MB)
- ✓ static/js/815.7e3a0d5d.js (3.66 MB) Data 页 echarts 相关 node_modules
- ✓ static/js/873.879e333e.js (3.29 MB)
- ✓ static/js/527.c03a10e2.js (1.46 MB) 编辑器相关 node_modules
- ✓ static/js/comps.70c7c27e.js (1.32 MB)
- ✓ static/js/537.9b13d9ea.js (1.06 MB)
- ✓ static/js/polyfill.38f619d5.js (973.97 KB)
- ✓ static/js/846.5df9fa4d.js (759.17 KB)
- ✓ static/js/990.c589e1c3.js (161.18 KB)
- ✓ static/js/main.db305dbb.js (125.43 KB)
- ✓ static/js/Publish.541f5592.js (83.68 KB) 发文页
- ✓ static/js/341.03dfcf08.js (75.49 KB)
- ✓ static/js/26.ada559b2.js (75.28 KB)
- ✓ static/js/Chapters.873cea8b.js (75.2 KB) 章节页
- ✓ static/js/328.3c207ce9.js (70.1 KB)
- ✓ static/js/55.f796d3f6.js (56.19 KB)
- ✓ static/js/513.cf0d6231.js (56.11 KB)
- ✓ static/js/710.19fd6744.js (54.84 KB)
- ✓ static/js/Payments.0d2c3d86.js (54.53 KB)
- ✓ static/js/587.f0af8f40.js (53.52 KB)
- ✓ static/js/443.68bc83b0.js (48.96 KB)
- ✓ static/js/Home.e5ede3c2.js (36.97 KB)
- ✓ static/js/Data.2147a217.js (32.58 KB) Data 页
- ✓ static/js/BookInfo.aa001ab1.js (31.67 KB)
- ✓ static/js/535.ca2d27ab.js (20.03 KB)
- ✓ static/js/Sign.76159403.js (19.13 KB)
- ✓ static/js/UserInfoEdit.eca71b89.js (17.83 KB)
- ✓ static/js/PaymentTerm.f639188b.js (14.49 KB)
- ✓ static/js/Books.446ccc23.js (14.31 KB)
- ✓ static/js/Notification.165c136d.js (13.06 KB)
- ✓ static/js/runtime.82480077.js (5.0 KB)

但是 Data 页整体 JS 包体积比发文页更大，FCP 却更小。

代码分析

国际化作家后台是一个客户端渲染的 SPA，它的整体代码结构如下：

```
1 const Content: React.FC = () => {
2   const { pathname } = useLocation();
3   const isImmersive = checkImmersive(pathname);
4
5   // 处于白名单的页面不需要添加header、affix、footer
6   const switchDomInWhitelist = (
7     <Switch>
8       {routers.map((router, index) => (
9         <Route {...router} key={index} />
10      ))}
11     <Redirect from="*" to="/" />
12   </Switch>
13 );
14
15   const switchDomInNormal = (
16     <>
17       <Header />
18       <div className="content">
```

```

19     <Affix
20         offsetTop={124}
21         style={{
22             left: '-170px',
23         }}
24         onChange={affixed => {
25             document.querySelectorAll<HTML<Element>(".nav")
[0].style.left = affixed ? '-180px' : '0';
26         }}
27     >
28     <Nav />
29 </Affix>
30     {switchDomInWhitelist}
31 </div>
32 <Footer />
33 <RightBar />
34 </>
35 );
36
37 return <>{isImmersive ? switchDomInWhitelist : switchDomInNormal}</>;
38 };

```

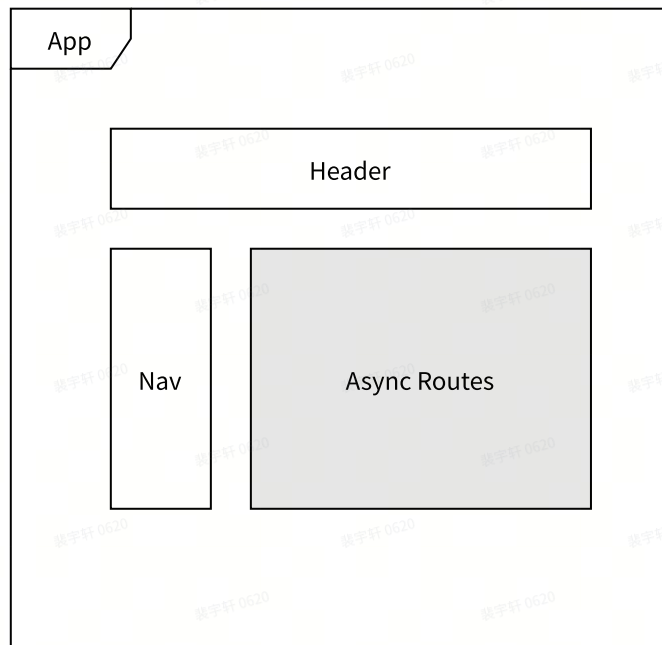
整体结构如下：

- Header：页面头部
- Nav：页面左侧导航栏
- Async Routes：懒加载的代码分割后的各个页面。在子页面记载过程中，会用一个 SVG Loading 元素进行渲染。

Loading 元素：

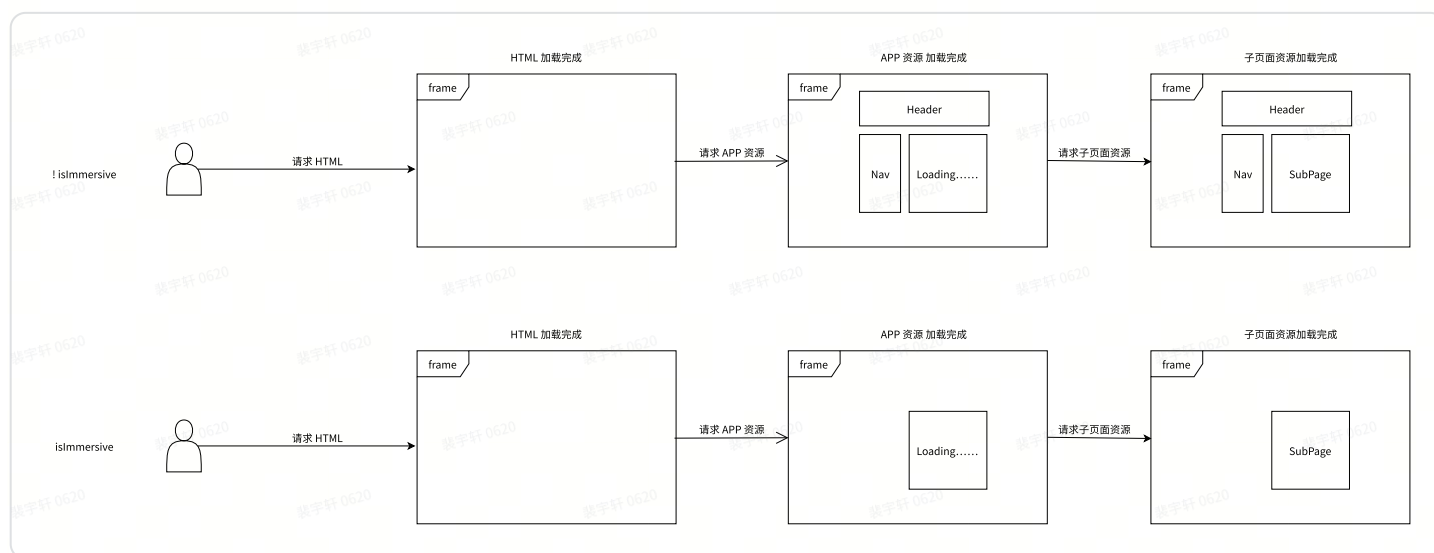


其中有三个页面比较特殊：publish/redirect/payment-term，也就是代码里的 isImmersive。这三个页面不会显示 Header 和 Nav



页面加载过程分析

所以页面整体的加载流程大概如下，按照 isImmersive 进行区分：



我们将页面加载分为 4 个阶段：

1. 用户发起请求
2. HTML 资源加载完成
3. APP 资源加载完成
4. 子页面资源加载完成

HTML 资源加载和 APP 资源加载是不受子页面影响的，也就是说对于所有页面来说，在相同的情况下，APP 资源加载完成这一步所需的时间应该是相同的。

用户发起请求 -> HTML 资源返回 -> APP 资源返回的耗时是固定的，现在我们看 APP 资源返回后页面的呈现情况：

- HTML 资源返回后页面是空白的，这时候页面并未触发 FCP
- APP 资源返回后
 - isImmersive 的子页面开始异步加载子页面资源，同时显示 Loading 元素
 - 非 isImmersive 的子页面也开始异步加载子页面资源，同时显示 Header、Nav 和 Loading 元素

所以到了 APP 资源加载完成的阶段，不管是否 isImmersive，页面都有内容渲染出来。isImmersive 页面只有 Loading 元素。

Loading 算 FCP 吗？

那么 Loading 元素渲染出来后算 FCP 吗？

情况一：算

如果 Loading 元素渲染算 FCP，那么所有页面的 FCP 应该一致才对，因为 FCP 是不受子页面加载影响的。

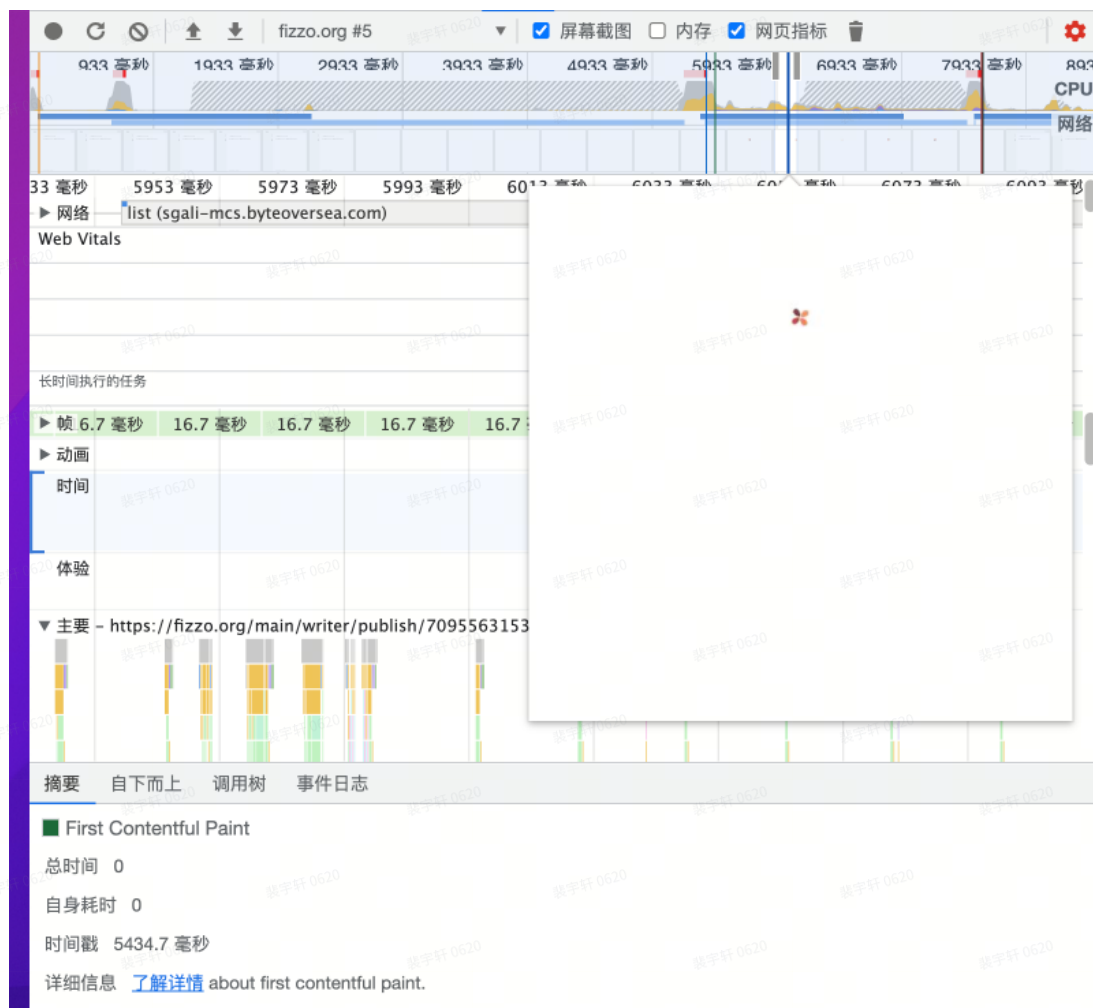
情况二：不算

如果 Loading 元素渲染不算 FCP，那么可以分析出了发文页 FCP 整体较长的原因：**因为其他页面没有算子页面加载，而发文页算上了子页面加载。**

按照 FCP 的定义，Loading 显然是算 FCP 的。

首次内容绘制 (FCP) 指标测量页面从开始加载到页面内容的任何部分在屏幕上完成渲染的时间。对于该指标，"内容"指的是文本、图像（包括背景图像）、`<svg>` 元素或非白色的 `<canvas>` 元素。

我们本地跑一下性能分析工具，看一下就知道了：



结果发现，FCP 时间为 5434ms，而在 6000ms 左右页面还是只有 Loading 元素。

所以结论是：**Loading 元素渲染算 FCP。**

但是根据前边的分析，如果 Loading 元素算 FCP，所有页面的 FCP 不都应该一致了吗？为什么会出现不同页面数据不一样的情况呢？

脚本测试

由于影响页面加载的因素很多，不同的设备、网络情况、国家地区等都可能影响页面加载。所以我们使用自动化测试来测试在固定的网络/设备等情况下，不同页面的 FCP 数据。

脚本

code 地址：<https://code.byted.org/liyan.2718/i18n-perf>

```
1 const puppeteer = require('puppeteer');
2
3 const fast3g = puppeteer.PredefinedNetworkConditions['Fast 3G'];
4 const slow3g = puppeteer.PredefinedNetworkConditions['Slow 3G'];
5
6 // ===== config =====
7 /**
```

```

8  * 需要测试的 URL
9  * { pid: url }
10 */
11 const UrlList = {
12     // 我们添加一个外网网站 YouTube 来与我们的网站性能进行对比
13     youtube: 'https://youtube.com',
14     home: 'https://fizzo.org/main/writer/home',
15     books: 'https://fizzo.org/main/writer/books',
16     publish: 'https://fizzo.org/main/writer/publish/7095563153265658881',
17     redirect: 'https://fizzo.org/main/writer/redirect',
18     paymentTerm: 'https://fizzo.org/main/writer/payment-term',
19 };
20
21 /**
22  * 网络或设备性能设置
23  * {
24  *     conditonName: {
25  *         cpu?: number; // CPU 慢 number 倍
26  *         net?: puppeteer.NetworkConditions; // 网络设置
27  *     }
28  * }
29 */
30 const Conditions = {
31     unlimit: {},
32     slowCpu4x: { cpu: 4 },
33     fast3g: { net: fast3g },
34 };
35
36 /**
37  * 每个页面跑几次测试
38  */
39 const COUNT = 10;
40 // ===== end config =====
41
42 async function getFCP(url, cond) {
43     const browser = await puppeteer.launch({ headless: true });
44     try {
45         const page = await browser.newPage();
46
47         // 设置登录态
48         await page.setCookie({
49             domain: 'fizzo.org',
50             name: 'sessionid',
51             value: '2f83220917f67e3f1705babba954ccf9',
52         });
53
54         // 设置网络性能

```

```
55     if (cond.net) {
56         await page.emulateNetworkConditions(cond.net);
57     }
58
59     // 设置 CPU 性能
60     if (cond.cpu) {
61         await page.emulateCPUThrottling(cond.cpu);
62     }
63
64     // 用来检查 Slardar 上报的数据是否和这里测试出来的一致，经过测试是一致的
65     // await page.setRequestInterception(true);
66     // page.on('request', interceptedRequest => {
67     //     if (interceptedRequest.isInterceptResolutionHandled()) return;
68     //     if (interceptedRequest.url().includes('/batch')) {
69     //         const data = JSON.parse(interceptedRequest.postData() ||
70     //             '{}');
71     //         data?.list?.forEach(item => {
72     //             if (item.ev_type === 'performance' &&
73     //                 item.payload?.name === 'fcp') {
74     //                 console.log(`Slardar fcp: ${item.payload.value}`);
75     //             }
76     //         });
77     //     }
78     //     interceptedRequest.continue();
79     // });
80
81     // 等待页面加载至可以测量 FCP
82     const navigationPromise = page.waitForNavigation({
83         waitUntil: 'networkidle2',
84         timeout: 60000,
85     });
86     await page.goto(url);
87
88     await navigationPromise;
89
90     let firstContentfulPaint = JSON.parse(
91         await page.evaluate(() =>
92             JSON.stringify(performance.getEntriesByName('first-contentful-paint'))
93         ));
94
95     const fcp = firstContentfulPaint[0].startTime;
96     return fcp;
97 } catch (error) {
98     console.log('error', error);
99 } finally {
100     await browser.close();
101 }
```



```

99 }
100
101 async function main() {
102     const stats = {};
103
104     const conds = Object.keys(Conditions);
105     const pids = Object.keys(UrlList);
106
107     for (let cond of conds) {
108         console.log(`Start measure condition ${cond}`);
109         let netStat = {};
110         for (let pid of pids) {
111             let count = COUNT;
112             const runs = [];
113             while (count-->0) {
114                 console.log(`Measure ${pid} at ${cond}`);
115                 const fcp = await getFCP(UrlList[pid], Conditions[cond]);
116                 runs.push(fcp);
117                 console.log({ fcp });
118             }
119             netStat[pid] = netStat[pid] || {};
120             const runsHasData = runs.filter(Boolean);
121             netStat[pid].runs = runsHasData;
122             netStat[pid].avg = runsHasData.reduce((a, b) => a + b, 0) /
runsHasData.length;
123             netStat[pid].p70 = runsHasData.sort((a, b) => a - b)
[Math.floor(runsHasData.length * 0.7)];
124
125             console.log({ [`${cond}-${pid}`]: netStat[pid] });
126         }
127         stats[cond] = netStat;
128         console.log({ [cond]: netStat });
129         console.log('-----');
130     }
131
132     console.log(JSON.stringify(stats, null, 4));
133 }
134
135 main();
136

```

测试

MAC, 家庭网络+seal, 20 runs/page

首先使用家庭网络+Seal 来进行测试, 每个页面跑 20 次。

配置：

```
1 const UrlList = {
2     google: 'https://google.com',
3     home: 'https://fizzo.org/main/writer/home',
4     publish: 'https://fizzo.org/main/writer/publish/7095563153265658881',
5     books: 'https://fizzo.org/main/writer/books',
6 };
7
8 const Conditions = {
9     unlimit: {},
10    slowCpu4x: { cpu: 4 },
11    fast3g: { net: fast3g },
12 };
13
14 const COUNT = 20;
```

原始数据：[📄 性能数据](#)

		google	home	publish	books
unlimit	avg	943	2153	2220	2178
	p70	1024	2482	2630	2574
slowCpu4x	avg	1004	3032	2656	2992
	p70	1143	3078	2738	3179
fast3G	avg	1574	7368	7282	7260
	p70	1589	7484	7439	8027

解读：

- 在不同的网络环境下，作家后台三个页面的 FCP 值都相差不大：FCP 与 PID 关联性较弱
- 在 slowCpu4x 的情况下，publish 页面平均值相比其他页面更小，应该是轮次较少数据方差大引起的，重复运行也有可能其他页面数值较小
- CPU 性能下降 4 倍使作家后台三个页面的 FCP 增长了 40% - 50%
- 使用 3g 网络使作家后台三个页面的 FCP 增长了 > 350%
- Google 主页整体性能在各个条件下都优于作家后台，且随 CPU/网络变化整体上升幅度更小。

MAC，公司网络，10 runs/page

根据上文分析，redirect 和 payment-term 这两个页面和 publish 的加载顺序类似。而且这两个页面都非常简单：

```
1 const Redirect: React.FC = () => <div>Redirecting, please wait</div>;
2
3 const PaymentTerms: React.FC = () => {
4   useEffect(() => {
5     window.scrollTo(0, 0);
6   }, []);
7
8   return (
9     <div className="payment-terms">
10       <Header />
11       { /* Policy 可以视为一个静态组件，主要是条款相关 */ }
12       <Policy showFooter={true} />
13     </div>
14   );
15 };
```

因此我们这次新加上这两个页面的测试。而且 Google 首页相对比较简单，我们这次取一个较大的外部应用来作为对比：YouTube。

配置：

```
1 const UrlList = {
2   youtube: 'https://youtube.com',
3   home: 'https://fizzo.org/main/writer/home',
4   publish: 'https://fizzo.org/main/writer/publish/7095563153265658881',
5   books: 'https://fizzo.org/main/writer/books',
6   redirect: 'https://fizzo.org/main/writer/redirect',
7   paymentTerm: 'https://fizzo.org/main/writer/payment-term',
8 };
9
10 const Conditions = {
11   unlimit: {},
12   slowCpu4x: { cpu: 4 },
13   fast3g: { net: fast3g },
14 };
15
16 const COUNT = 10;
```

原始数据：[📄 性能数据](#)

		youtube	home	publish	books	redirect
unlimit	avg	1509	4013	3573	4391	
	p70	1643	4525	3614	4501	
slowCpu4x	avg	1153	4922	4604	4594	
	p70	1225	4925	4902	4794	
fast3G	avg	3138	8688	7392	7622	
	p70	3297	7716	7570	7778	

解读：

- 整体趋势仍然符合第一轮次的测试结果
- redirect 和 payment-term 这两个页面，子页面非常简单，但是 FCP 数据相比其他页面仍然没有明显的下降

MAC, PPE, 公司网络, 10runs/page: 测试编辑器相关包的影响

为了验证编辑器相关包对发文页的影响，我们在 PPE 添加 publish-noeditor 路由，将 publish 页面编辑器相关的代码移除掉，其他仍然保留。

commit: <https://code.byted.org/novel-fe/serial-author-web-i18n/commit/54e08906aa40d1f7ccb4ed7061eff0f777b55374>

发布 PPE 后，测试 Home 页、发文页、去掉编辑器的发文页的 FCP 数据。

脚本设置：

```

1 const UrlList = {
2   home: 'https://fizzo.org/main/writer/home',
3   publish: 'https://fizzo.org/main/writer/publish/7095563153265658881',
4   publishNoEditor: 'https://fizzo.org/main/writer/publish-noeditor/7095563153265658881',
5 };
6
7 const Conditions = {
8   unlimit: {},
9   slowCpu4x: { cpu: 4 },
10  fast3g: { net: fast3g },
11 };
12
13 const COUNT = 10;

```

页面设置 PPE:

```
1 await page.setRequestInterception(true);
2 page.on('request', interceptedRequest => {
3   if (!interceptedRequest.isNavigationRequest()) {
4     interceptedRequest.continue();
5     return;
6   }
7   const headers = interceptedRequest.headers();
8   headers['x-use-ppe'] = '1';
9   headers['x-tt-env'] = 'ppe_i18n_perf';
10  interceptedRequest.continue({ headers });
11 });
```

原始数据: [性能数据](#)

		home	publishNoEditor	publish	备注
unlimit	avg	2442	2605	2282	
	p70	2571	2375	2414	
slowCpu4x	avg	3320	3604	3425	
	p70	3739	4095	3895	
fast3G	avg	7605	6903	6948	home 去掉一个异常值 avg 为 6931
	p70	7045	6980	7070	

解读:

- 整体趋势仍然符合第一轮测试: FCP 与 PID 关联性较弱
- 去掉编辑器 SDK 后, 发文页 FCP 与未去掉的情况下相比无大的差别。小的差别可以认为是方差大引起的。

Windows, 公司网络, 10runs/page

配置:

```
1 const UrlList = {
2   youtube: 'https://youtube.com',
3   home: 'https://fizzo.org/main/writer/home',
4   publish: 'https://fizzo.org/main/writer/publish/7095563153265658881',
5   books: 'https://fizzo.org/main/writer/books',
```

```

6     redirect: 'https://fizzo.org/main/writer/redirect',
7     paymentTerm: 'https://fizzo.org/main/writer/payment-term',
8 };
9
10 const Conditions = {
11     unlimit: {},
12     slowCpu4x: { cpu: 4 },
13     fast3g: { net: fast3g },
14 };
15
16 const COUNT = 10;

```

原始数据：[性能数据](#)

		youtube	home	publish	books	redirect
unlimit	avg	1105	2359	2317	2462	
	p70	1137	2373	2426	2548	
slowCpu4x	avg	1332	3191	3360	3245	
	p70	1372	3255	3443	3312	
fast3G	avg	3573	7091	7058	7104	
	p70	3663	7151	7111	7088	

解读：

- 数据表现和前边三轮测试一致

Slardar 用户分析

- 根据上文测试发现，在其他条件相同的情况下，不同 pid 的 FCP 基本一致。那么 Slardar 上报的 FCP 为什么会不一样呢，特别是发文页总是比其他页面有更高的指标值？是否是因为不同页面的用户设备/网络等情况不一样？

我们使用 Slardar 的用户分析来查看home/books/publish三个页面的数据情况：

数据探索，11月-1日 到 11月-10日的 PV 统计：https://slardar-us.bytedance.net/node/web/data_search?env=Slardar_All&bid=novel_author_i18n&lang=zh&start_time=1667232000&end_time=1668069229&site_type=web&layout=normal&ev_type=view&filter_id=0d158ec4e621e79efc4a8d6699fb167a&granularity=auto

PV 网络分布：


全部：

网络类型	home		books		publish	
	计数	百分比	计数	百分比	计数	百分比
4g	157420	73.62198464	92930	72.28587652	134870	67.92
3g	39930	18.67441143	24590	19.12740454	44130	22.224
空	13400	6.266894894	9250	7.195139975	17460	8.793
slow-2g	961	0.449439253	554	0.430930546	820	0.4129
2g	821	0.383964232	465	0.361701631	710	0.3575
wifi	434	0.202972566	344	0.267581422	285	0.1435
cellular	831	0.388641019	419	0.325920395	280	0.14
none	25	0.011691968	7	0.005444971	6	0.0030
总计	213822		128559		198561	

有 FCP 的：

网络类型	home		books		publish	
	计数	百分比	计数	百分比	计数	百分比
4g	62250	76.51838285	12880	75.61791816	4420	73.678
3g	13700	16.84019028	3040	17.84770739	1140	19.003
空	4660	5.728123118	896	5.260376915	344	5.7342
slow-2g	387	0.475704645	116	0.68103094	58	0.9668
2g	309	0.37982619	85	0.499031292	31	0.5167
wifi	33	0.040563962	1	0.005870956	0	
cellular	14	0.017208954	15	0.088064346	6	0.1000
none	0	0	0	0	0	
总计	81353		17033		5999	

解读：

- 
- publish 页面的 3g 用户占比显著高于其他两个页面
 - home页有 FCP 数据的 PV 占有所有 PV 的38%，books 页为12%，publish 页为 3%（平均每天 ~600PV）
 - 说明 publish 页主要来源是通过 home/books 直接当前页面跳转过去，这种情况下不会产生 FCP

PV 设备分布：


全部：

设备	home		books		publish	
	计数	百分比	计数	百分比	计数	百分比
Windows	81100	38.02976732	59220	46.27430143	121700	61.310
Android	62310	29.2186782	30170	23.57473276	26170	13.184
GNU/Linux	61950	29.04986542	32760	25.59854973	40270	20.287
Mac	3650	1.711573992	3060	2.391073326	5250	2.6448
iOS	2290	1.073836833	1670	1.304932175	2840	1.4307
Chrome OS	1890	0.886267081	1080	0.843908233	2200	1.1083
Ununtu	49	0.022977295	12	0.009376758	65	0.0327
其他	15	0.007033866	4	0.003125586	2	0.0010
总计	213254		127976		198497	

有 FCP 的：

设备	home		books		publish	
	计数	百分比	计数	百分比	计数	百分比
Windows	28570	35.11165186	5310	31.18942731	1290	
Android	25790	31.69511731	5040	29.60352423	1590	
GNU/Linux	24100	29.61815925	5870	34.47870778	2810	46.833
Mac	1400	1.720556969	412	2.419970631	153	
iOS	916	1.125735845	300	1.762114537	107	1.7833
Chrome OS	591	0.726320835	93	0.546255507	44	0.7333
Ununtu	2	0.002457939	0	0	6	
none	0	0	0	0	0	
总计	81369		17025		6000	

解读：

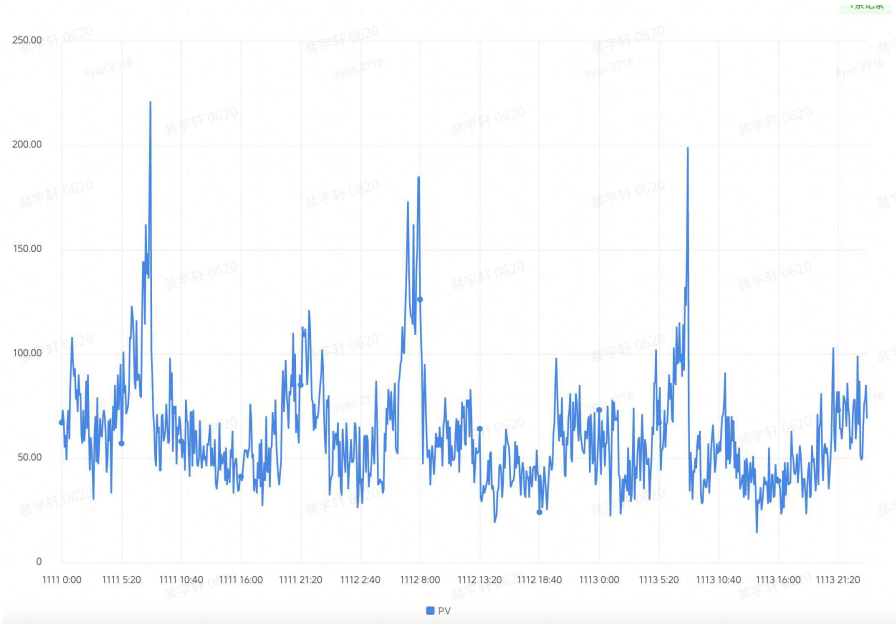
- 
- publish 页面 windows 用户 PV 占比远高于其他两个页面，但是有 FCP 上报的 PV 中，windows 用户远低于其他页面，说明直接使用 windows 系统通过 books/home 进入 publish 页面占publish 页面 PV 的大多数，而这部分用户不上报 FCP
 - publish 页面的 FCP 上报主要为非 windows 设备，推测作者可能是将发文链接存为书签，然后通过非桌面设备在 3g 网络下进行创作
 - publish 页面有每日仅有不到 600 的 FCP 上报，而 home 页为 >8000

PV 影响

如果 FCP 主要是受设备以及网络类型的影响，我们可以固定国家、设备、网络等条件，查看不同 PID 下的 FCP 数据。

不过在尝试了各种时间粒度以及网络情况后，发现发文页通常会有较大的指标值。

发文页有 FCP 上报的 PV 每天仅约 600 次，平均到每个小时只有 24 次，经过网络条件筛选后可能更少。因此我们取一个发文页 PV 的高峰期来进行观察。



观察发现每天 7-8 点是发文页的 PV 高峰。（不过为什么大早上是高峰期呢，印尼时区和北京时区差不多，印尼人起这么早吗？）

数据：https://slardar-us.bytedance.net/node/web/multi-analyze?env=Slardar_All&bid=novel_author_i18n&lang=zh&start_time=1667232000&end_time=1667318399&site_type=web&id=daef8955e317beb717d59fdf384ab883&layout=normal&container_type=undefined

FCP（单位：s）

	home	books	publish
11月14日	2.79	2.58	3.84
11月13日	2.95	4.79	2.85
11月12日	3.42	4.36	2.77
11月11日	3.48	3.16	3.57
11月10日	3.48	4.93	2.66
11月9日	2.88	2.87	2.46
11月8日	2.65	3.44	2.4
平均	3.092857143	3.732857143	2.935714286

解读：



- 高峰期并未观察到发文页 FCP 比其他页面指标高的现象

初步结论

- 对于作家后台项目，其他条件都相同的情况下，不同页面的 FCP 是趋于一致的
- 子页面资源包大小对 FCP 基本无影响（因为都是异步加载，加载过程中已经触发了 FCP）
- 网络性能对页面 FCP 影响巨大
- 不同 PID 的 FCP 数据不一致，更多是受不同页面设备、网络性能分布不同影响
- PV 较少有可能影响统计数据的准确性
- 不同网络/设备性能条件下，作家后台性能与其他项目（Google/YouTube）相比，性能均严重落后，特别是 3g 网络下落后更明显

团队现在性能数据按照同一站点不同 PID 的 FCP 来进行统计是不准确的

FCP 适合对比不同站点的，或者相同站点但不是作家平台这种子路由架构的

这种异步子路由的页面更适合用 LCP 来进行度量

诸如load（加载）或DOMContentLoaded（DOM 内容加载完毕）这样的旧有指标并不是很好，因为这些指标不一定与用户在屏幕上看到的内容相对应。而像First Contentful Paint 首次内容绘制 (FCP)这类以用户为中心的较新性能指标只会捕获加载体验最开始的部分。如果某个页面显示的是一段启动画面或加载指示，那么这些时刻与用户的关联性并不大。

二：整体 FCP 较慢分析

为了模拟真实的海外用户，我们采用<https://www.webpagetest.org/>平台来对国际化作家平台进行网站性能测试。

Start a **Site Performance** Test!

<https://fizzo.org/main/writer/home>

Simple Configuration 3 test runs from recommended location and browser presets

Advanced Configuration Choose from all browser, location, & device options

Test Location: **Jakarta, Indonesia - GCE** 地区 选择印度尼西亚

Browser: **Chrome**

Start Test →

Test Settings	Advanced	Chromium	Script	Block	SPOF	Custom
Connection	4G (9 Mbps, 170ms RTT) <small>选择网络类型</small>					
Desktop Browser Dimensions	default (1366x768)					
Number of Tests to Run	1					
Repeat View	<input checked="" type="radio"/> First View and Repeat View			<input type="radio"/> First View Only		
<input checked="" type="checkbox"/> Capture Video						

*由于 webpagetest 测试没添加上登录态，页面发生了一次跳转。测试中的 Start Render 时间为实际有内容渲染出来，我们主要关注从开始到 Start Render 之间的性能数据。

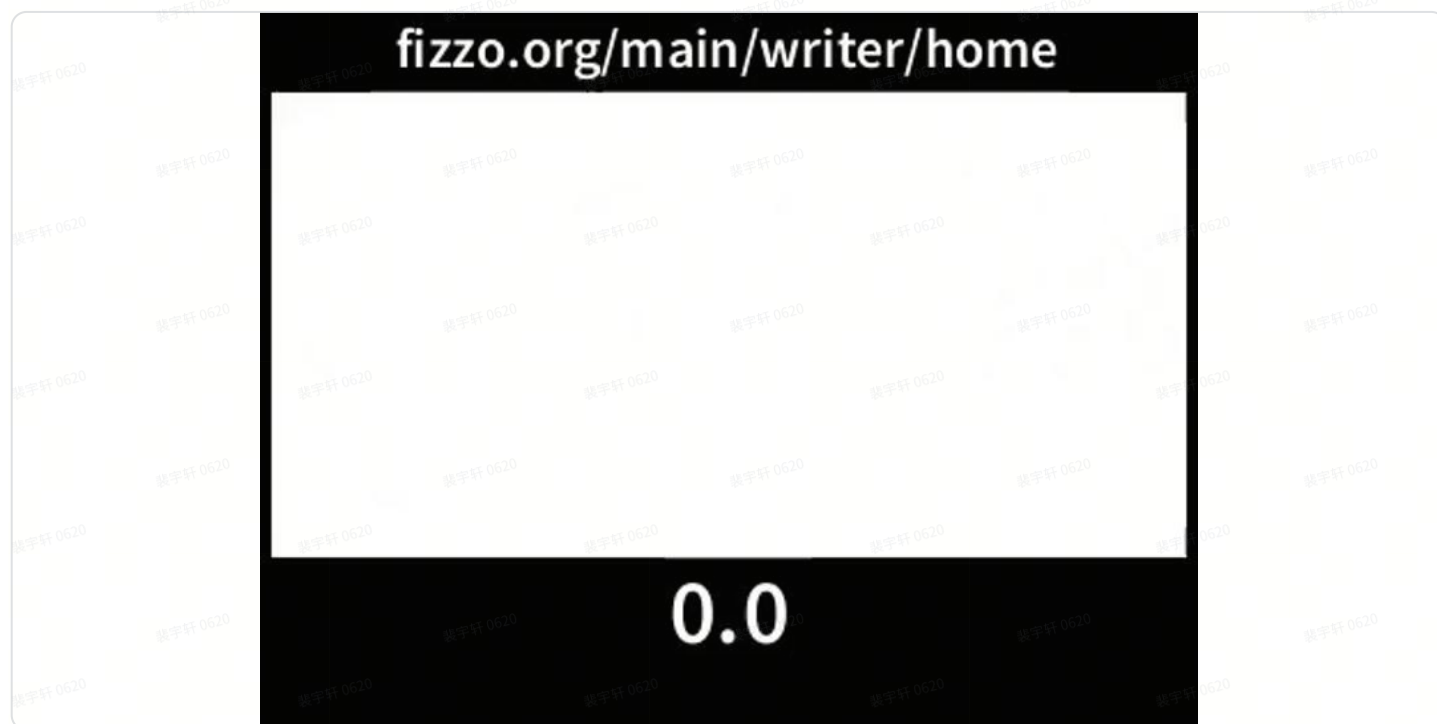
Webpagetest 测试结果

4G

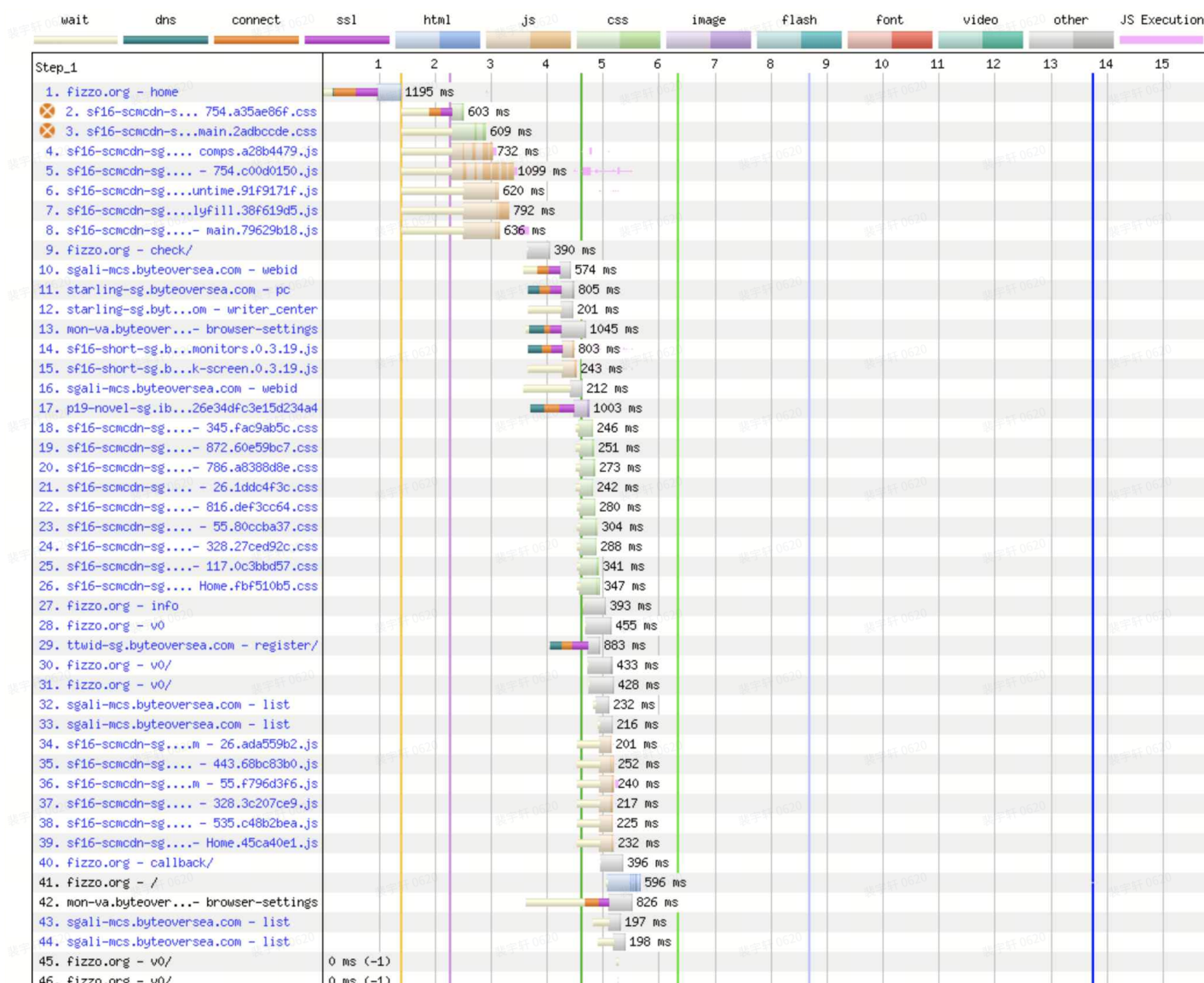
结果：

https://www.webpagetest.org/result/221117_AiDc22_4CT/1/details/#waterfall_view_step1

视频：



时间线：



Fast 3G

结果:

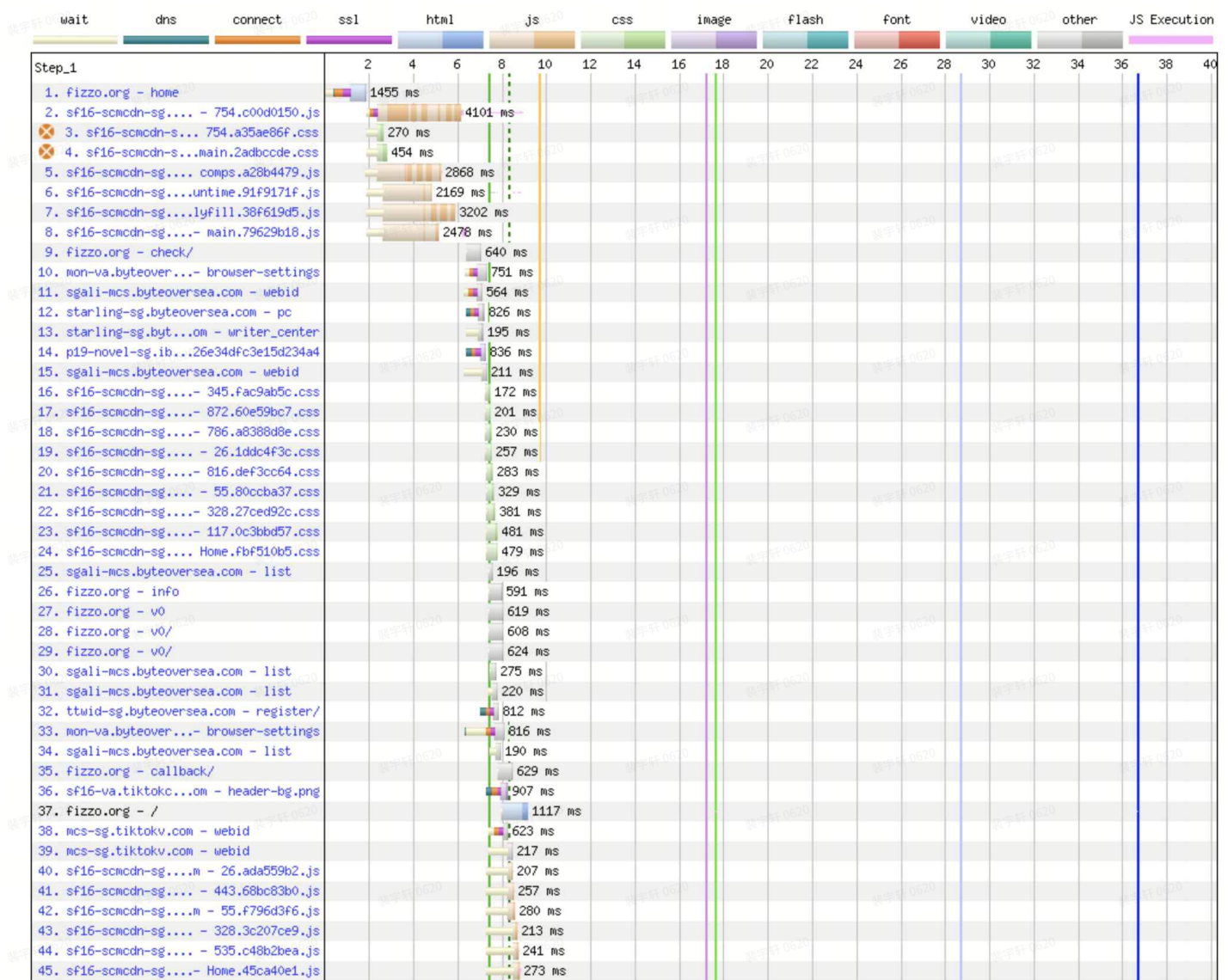
https://www.webpagetest.org/result/221117_AiDcA7_4CW/1/details/#waterfall_view_step1

视频:

fizzo.org/main/writer/home

0.0

时间线:

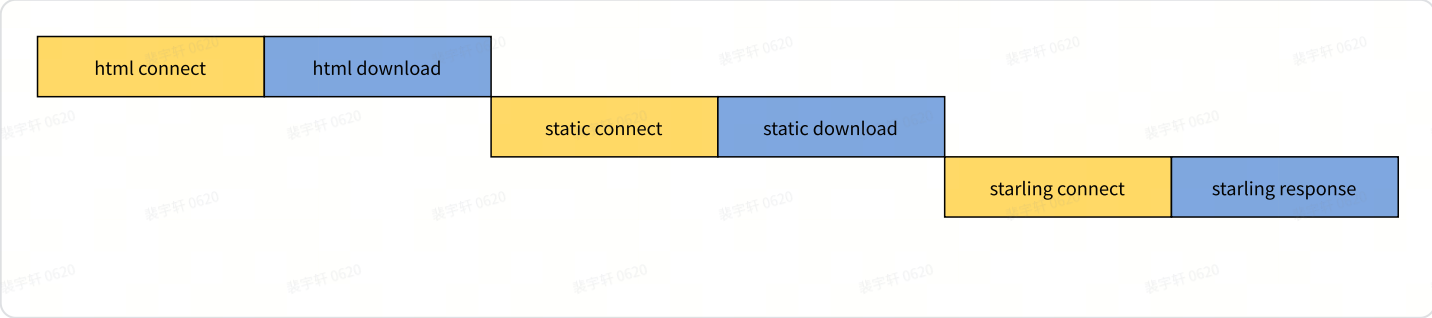


分析

页面加载主要分为以下几个步骤:

- 1. 加载页面 HTML，域名：fizzo.org
- 2. 解析 HTML，并加载JS/CSS静态文件，域名：sf16-scmcdn-sg.ibytedtos.com
- 3. JS 加载完成，发起翻译平台接口调用，域名：starling-sg.byteoversea.com
- 4. 翻译平台接口返回，React 渲染展示页面

整体流程如下：



耗时（ms）统计如下：

网络	html 连接时间	html 下 载时间 (等待时间+下载时间)	js/css 连接时间	js/css 下载时间	starling 连接时间	starling 返回时间	建立连接时间	下载资源时间	连接 + 下载总时间
4G	769	426 (421 + 5)	393	1099 (211+888)	574	(203+28)	1736	1756	3492
3G	781	674 (672+2)	337	3764 (468+3296)	530	296 (201+95)	1648	4438	6086

- 三个资源（HTML、静态资源、Starling）分别处在三个域名下，每个域名都需要首先建立网络连接，耗费了大量时间。4G 网络下建立连接时间为 1736ms，3G 下指标接近，为 1648ms。
- 3G 网络下，静态资源下载占了 3764ms 时间。

优化思路：

- 1. 缩短网络连接时间
- 2. 减少 JS/CSS 资源包体积，降低资源下载时间

减少网络连接时间

preconnect

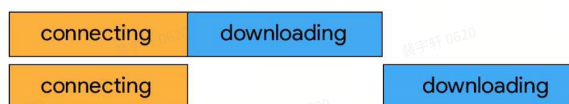
减少网络连接时间可通过 preconnect 提前建立连接来实现：

```
<link rel="preconnect" href="https://example.com">  
<link rel="preconnect" href="https://cdn.example.com">
```

Without preconnect



With preconnect



分别分析初始加载的三个网络连接：

- HTML：为初始请求，无法干涉
- JS/CSS静态资源：
 - 由于 JS/CSS 建立连接需要等待 HTML 返回，所以通过 `<link rel="preconnect">` 或者 `Link 返回头` 来建立提前连接是无效的。
 - 103 Early Hint：Chrome 103 版本开始支持 103 返回码，在正式请求返回前返回一个 103 状态码告诉浏览器预先连接一些资源。不过该特性目前在 nginx 还是实验特性，公司基础设施也不支持。
- Starling：可以通过 `<link rel="preconnect">` 来进行优化。

因此三个步骤中，我们能进行的优化就是提前建立到 Starling 的连接。

域名动态加速：

<https://tools.keycdn.com/performance?url=https://www.fizzo.org>

<https://tools.keycdn.com/performance?url=https://fizzo.org>

分析发现 fizzo.org 连接速度比 www.fizzo.org 慢：

URL					
<div>https://fizzo.org</div> <div>Test</div>					
LOCATION	STATUS	DNS	CONNECT	TLS	TTFB
Frankfurt	200	14.39 ms	9.07 ms	28.94 ms	226.92 ms
Amsterdam	200	10 ms	1.53 ms	21.36 ms	535.48 ms
London	200	10.23 ms	3.08 ms	13.26 ms	224.34 ms
New York	200	38.72 ms	35.6 ms	49.41 ms	381.99 ms
Dallas	200	12.83 ms	42.25 ms	53.97 ms	1.04 s
San Francisco	200	151.34 ms	141.09 ms	153.16 ms	662.79 ms
Singapore	200	4.3 ms	220.48 ms	229.37 ms	697.98 ms
Sydney	200	187.87 ms	194.77 ms	209.36 ms	827.4 ms
Tokyo	200	124.93 ms	104.03 ms	111.3 ms	526.97 ms
Bangalore	200	486.56 ms	149.44 ms	158.99 ms	1.46 s

URL					
<div>https://www.fizzo.org</div> <div>Test</div>					
LOCATION	STATUS	DNS	CONNECT	TLS	TTFB
Frankfurt	200	61.44 ms	0.57 ms	16.35 ms	241.57 ms
Amsterdam	200	24.85 ms	1.67 ms	18.9 ms	215.2 ms
London	200	17.86 ms	1.51 ms	16.7 ms	662.86 ms
New York	200	31.03 ms	3.18 ms	21.96 ms	1.39 s
Dallas	200	250.43 ms	1.08 ms	21.21 ms	850.08 ms
San Francisco	200	17.13 ms	1.66 ms	13.68 ms	758.68 ms
Singapore	200	6.17 ms	1.85 ms	13 ms	58.96 ms
Sydney	200	560.48 ms	1.5 ms	13.14 ms	509.23 ms
Tokyo	200	273.26 ms	0.87 ms	74.45 ms	454.29 ms
Bangalore	200	220.56 ms	2.3 ms	17.63 ms	306.95 ms

在 NetLink 上发现 fizzo.org 未开启域名动态加速。此处可以将 fizzo.org 页打开域名动态加速。

<input type="checkbox"/>	fictum.org	七层全链路接入	<div><div>域名</div><div>动态加速</div><div>TLB</div></div>	2022-07-05 23:26:59	<div>配置链路</div> <div>域名Tag</div> <div>迁移空间</div>
<input type="checkbox"/>	fizzo.org	七层全链路接入	<div><div>域名</div><div>动态加速</div><div>TLB</div></div>	2022-07-05 23:26:59	<div>配置链路</div> <div>域名Tag</div> <div>迁移空间</div>
<input type="checkbox"/>	www.fictum.org	七层全链路接入	<div><div>域名</div><div>动态加速</div><div>TLB</div></div>	2022-07-05 23:26:59	<div>配置链路</div> <div>域名Tag</div> <div>迁移空间</div>
<input type="checkbox"/>	www.fizzo.org	七层全链路接入	<div><div>域名</div><div>动态加速</div><div>TLB</div></div>	2022-07-05 23:26:59	<div>配置链路</div> <div>域名Tag</div> <div>迁移空间</div>

共 4 条 < 1 > 10 条/页

减少资源包体积

首次加载有 3 个 JS 文件和 2 个 CSS 文件。

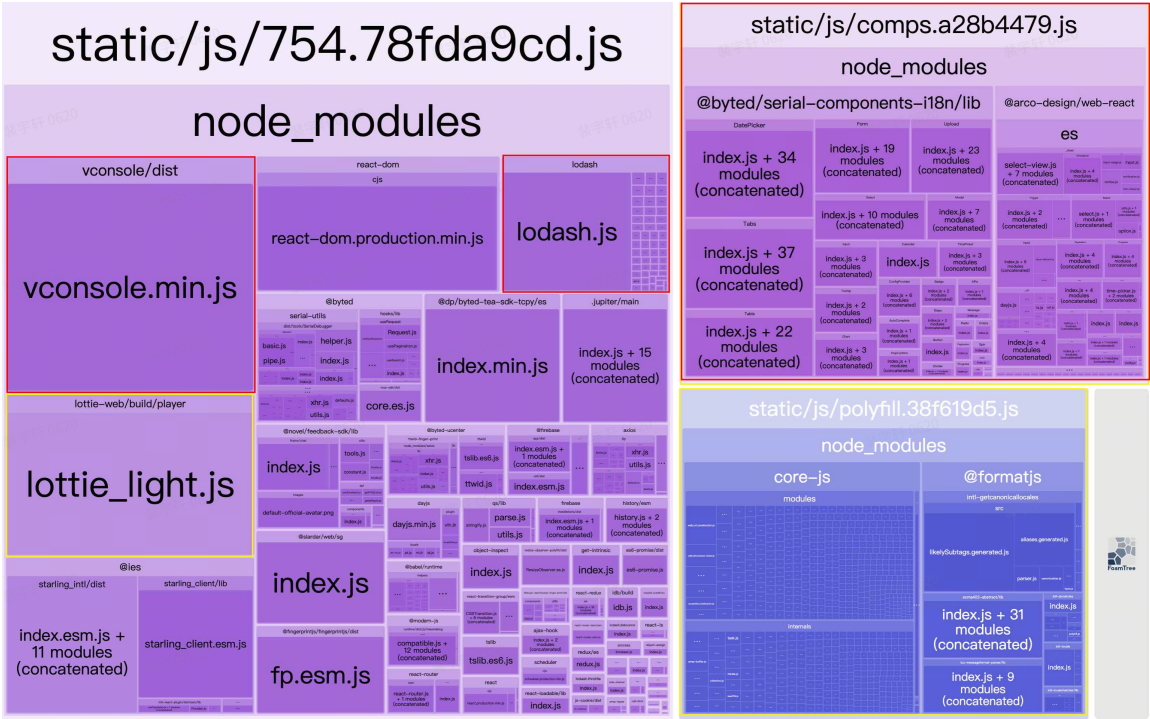
3 个 JS 文件共 655.31KB，分别为：

- App：394.75KB

- 组件库：148.13KB
- Polyfill (Corejs + formatjs)：112.43KB

组件库和 CoreJS 平时更改较少，分包可以优化缓存。

分析 Webpack 打包结果



1. 确定可以优化的：

- vconsole：65.73KB。debugger 工具，可以异步加载
- lodash：25.55KB。可以按需加载

2. 对比后可以优化的：

- 组件库

分包：单独构建一个资源包，在版本更新中保持 hash 不变，利于缓存。

不分包：组件库分散到各个子页面内，子页面修改后这部分组件无缓存需要重新加载。

组件库进行分包的好处是有益于缓存，但是也带来了一些问题：一些子页面会加载不需要的组件。

	分包	不分包
全新访问	初始请求加载大量不需要组件，对 FCP 有影响	减少组件库初始代码，页面呈现更快。子页面新增了少量组件库代码，LCP 会少量上升。但综合初始请求，总 LCP 应该下降。
页面完全缓存	直接加载缓存，大小无影响	直接加载缓存，大小无影响
部分更新	<ul style="list-style-type: none">• APP部分JS重新加载	<ul style="list-style-type: none">• APP部分JS重新加载

- | | | |
|--|--|--|
| | | <ul style="list-style-type: none">子页面 JS 重新加载，因为 JS 体积有少量增加，LCP 会有增加 |
|--|--|--|

如果版本更新不是太快，未命中缓存的情况将主要是因为全新访问。因此可以考虑将组件库取消分包。

在 PPE 测试中，取消分包后在 fast3G 网络下 FCP 下降了约 500-600ms。

可以调研后确定是否优化的：

- lottie: 45.29KB。Lottie 的作用是展示 Loading SVG 图标，图标比较小，但为了播放 Lottie 动画引入 lottie-web，占了初始资源的 7%。可以考虑不使用 Lottie 来实现 Loading 图标。
- core-js: 64.82KB。可以考虑进行按需加载。

优化操作

结合上述分析，先执行下列优化：

1. 预连接 Starling:
2. 异步加载 debugger
3. 按需加载 lodash
4. 取消组件库分包
5. fizzo.org 开启域名动态加速

优化后打包起始 JS 文件体积为 424.66KB:

- APP 313.23KB
- Polyfill 112.43KB

共减少 $655.31 - 424.66 = 230.65\text{KB}$ ，占比 35.2%。

预计节约加载时间：

4G: $0.352 * 1099 + 574 = 886.6\text{ms}$ ，占 FCP 的 19.3%

Fast3G: $0.352 * 3296 + 530 = 1690\text{ms}$ ，占 FCP 的 22.2%

Commit: https://code.byted.org/novel-fe/serial-author-web-i18n/merge_requests/293

进一步优化可能

- corejs 按需引入：预计可减少包体积 60.02 KB
- 移除 lottie：最大可减少 45.29KB

在 4G 和 fast3G 条件下可减少下载时间 176.9ms 和 530.7ms

- 进一步减少网络连接耗时，比如 103 early hint

PS：国家/地区对性能数据影响

国际化作家平台 PV 按国家统计

- 印尼占比 > 87%
- 印尼 + 委内瑞拉 + 尼日利亚 > 91%

添加到看板

导出

天级

2023-01-05 00:00:00 - 2023-01-12 00:00:00

...

指标

Q PV

+

分组

国家/地区

+

Top N

包含 Null 值

查询

自动

76条记录

分组指标	指标和	平均值	指标值	20230104	20230105	20230106
PV 印度尼西亚	474.07 K	59.26 K	474.07 K	61.25 K	59.05 K	62.43 K
PV 巴西	17.29 K	2.16 K	17.29 K	3.03 K	2.37 K	2.21 K
PV 委内瑞拉	12.21 K	1.53 K	12.21 K	3.05 K	1.46 K	1.20 K
PV 尼日利亚	9.68 K	1.21 K	9.68 K	1.64 K	1.05 K	943.00
PV 菲律宾	8.80 K	1.10 K	8.80 K	1.12 K	2.29 K	707.00
PV 哥伦比亚	4.90 K	612.50	4.90 K	1.26 K	821.00	717.00
PV 墨西哥	4.60 K	574.50	4.60 K	1.57 K	350.00	528.00

<

1

2

3

4

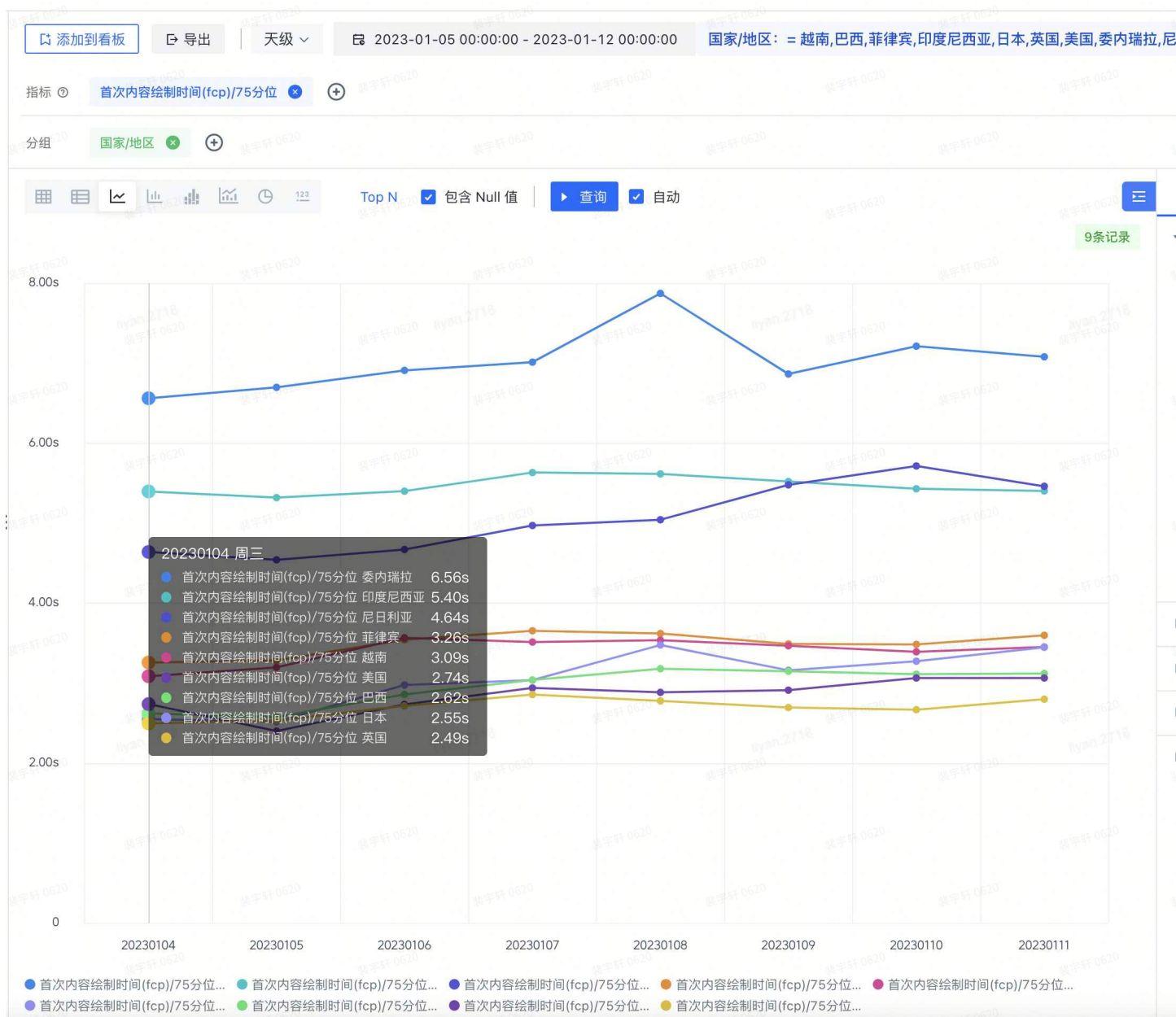
5

...

11

>

TikTok Web 版 FCP 数据



TikTok Web 版 TCP 连接速度



TikTok Web 版静态资源下载速度

指标 静态资源/下载速度/75分位

分组 国家/地区

Top N

包含 Null 值

查询

自动

